Python for Data Science & AI

Dasun Athukoralage

web: www.dasuna.me
email: dasun@nirvanaclouds.com

Python Tuples

Tuples are used to store multiple items in a single variable.

```
sports = ("tennis", "cricket", "football")
```

A tuple is a collection which is **ordered** and **unchangeable** and **allow duplicate values**.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Python Tuples

Note: Python tuples are immutable.

Tuple length:

To determine how many items a tuple has, use the len() function.

```
sports = ("tennis", "cricket", "football")

print(len(sports))
```

Python Tuples

Note:

If you want to create a tuple with single item, you have to add a comma after the item.

```
tuple_1 = ("tennis",)

print(type(tuple_1))

#NOT a tuple

tuple_1 = ("tennis")

print(type(tuple_1))  # this will be a string
```

Python Tuples

A tuple can contain different data types:

```
my_tuple = ("tennis", 30, True, "cricket", "football")
```

Negative indexing:

Like Python lists, you can use negative indexing to start accessing items from the end.

```
print(my_tuple[-2]) #output?
```

Python Tuples

You can specify a range of indexes by specifying where to start and where to end the range.

```
my_tuple = ("tennis", 30, True, "cricket", "football")

print(my_tuple[2:5])

print(my_tuple[:4])

print(my_tuple[2:])
```

Python Tuples

Update Tuples:

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

You can **convert** the tuple into **a list**, **change the list**, and **convert** the list **back into a tuple**.

Python Tuples

```
x = ("tennis", "cricket", "football")

y = list(x)

y[2] = "baseball"

x = tuple(y)

print(x)
```

Python Tuples

Similarly, if you want to **append** (i.e., add) an item or **remove** an item from a tuple, the workaround is to first convert it to a list, perform the required operations, and then convert it back to a tuple.

Python Tuples

Nested Tuples:

A nested tuple is simply a tuple where one or more elements are themselves tuples.

```
# Nested Tuple

nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))
```

Python Tuples

Nested Tuples:

```
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))

# Accessing elements in the nested tuple

print(nested_tuple[0])      # Output: (1, 2, 3)

print(nested_tuple[0][1])   # Output: 2

print(nested_tuple[1][2])   # Output: c

print(nested_tuple[2][0])   # Output: True
```

Python Sets

Python Sets

Sets are used to store multiple items in a single variable.

```
sports = {"tennis", "cricket", "football"}
```

A set is a collection which is **unordered** and **changeable (you can add or remove items but you can’t change or update an item that is already in a set)** and **does not allow duplicate values**.

Set items are not indexed, so you cannot access them by position like in lists or tuples.

Python Sets

```
sports = {"tennis", "cricket", "football"}

print(sports) #Output: {'cricket', 'football', 'tennis'}
print(type(sports)) #<class 'set'>
```

As you can see, the order of the output can’t be guaranteed.

Note: Python sets are mutable.

Python Sets

---

len() function

To determine how many items a set has, use the `len()` function.

```
this_set = {"apple", "banana", "cherry"}

print(len(this_set))
```

Python Sets

Property:

A set can contain different data types:

```
set_1 = {"abc", 34, True, 40, "male"}
```

Python Sets

Access Items

Once a set is created, **you cannot change its items**, but you can add new items (not like tuples). Therefore **Sets are mutable**.

Python Sets

Add items

Python sets are mutable in the sense that you can add new elements after creation using methods like **`.add()`** or **`.update()`**.

Python Sets

Add items

```
sports = {"tennis", "cricket", "football"}

sports.add("soccer")  # Adds a single item

print(sports)  # Output might be: {'tennis', 'cricket', 'football',
'soccer'}

sports.update(["basketball", "volleyball"])  # Adds multiple items

print(sports)  # Output might be: {'tennis', 'cricket', 'football',
'soccer', 'basketball', 'volleyball'}
```

Python Sets

Removing items

You can also remove items using methods like **.remove()** (raises an error if the item isn't found) or **.discard()** (no error if the item isn't present).

```
sports = {"tennis", "cricket", "football"}

sports.remove("cricket")

print(sports)  # Output might be: {'tennis', 'football'}
```

Python Range

Python Range

In Python, range() is a built-in function used to create an **immutable sequence of numbers** representing an arithmetic progression.

The range() function can be used to generate a range of integers based on a **start value, stop value, and step value**.

It is commonly used in **loops** and other scenarios where a sequence of numbers is needed.

Python Range

The syntax for the `range()` function is as follows:

```
range(start, stop, step)
```

- **start:** The starting value of the range (inclusive). If not provided, the range starts from 0.
- **stop:** The ending value of the range (exclusive). The range goes up to, but does not include, this value.
- **step:** The difference between each consecutive number in the range. If not provided, the default step is 1.

Python Range

```
my_range = range(1, 10, 2)  # Represents the sequence 1, 3, 5, 7, 9


item = my_range[2]  # Accessing the item at index 2 (5)


print(item)  # Output: 5
```

Mapping Data Type: Python Dictionaries

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is **ordered**, **changeable** and **does not allow duplicates**.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
print(this_dict)
```

Python Dictionaries

Dictionary Item

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Print the "price" value of the dictionary:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
print(this_dict[“price”])
```

Python Dictionaries

Duplicates Not Allowed

Dictionaries cannot have two items with the same key: **Duplicate values will overwrite existing values**:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35,
   "price": 128.15
}
print(this_dict)
```

To determine how many items a dictionary has, use the len() function:

```
print(len(this_dict))
```

Python Dictionaries

Dictionary items - Data types

Strings, Numbers, Booleans, Lists, Tuples, Sets, Other dictionaries, Custom objects:

```
my_dict = {
    "name": "John",
    "age": 30,
    "is_student": False,
    "grades": [85, 92, 78],
    "coordinates": (12.34, 56.78),
    "preferences": {"color": "blue", "food": "pizza"}
}
```

Dasun Athukorala

Python Dictionaries

Dictionary keys - Data types

Dictionary **keys must be immutable** (to have a constant hash value).

❌ **You cannot use the following as dictionary keys:**

- **list** (e.g., [1, 2])
- **dict** (e.g., {"a": 1})
- **set** (e.g., {1, 2, 3})

✅ **You can use these types as dictionary keys:**

- **int, float, str, bool**
- **tuple** (only if it contains only hashable types)
- **frozenset** (immutable version of set)

Python Dictionaries

Accessing items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Get the value of the "price" key.

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
```

x = this_dict["price"]

There is also a method called get() that will give you the same result:

x = this_dict.get("price")

Python Dictionaries

Change Values

You can change the value of a specific item by referring to its key name:

Change the "price" to 119.20:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict["price"] = 119.20
```

Python Dictionaries

Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Update the “name” and "price" by using the update() method:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict.update({"name": “Salt”, "price": 112.80})   # Output: {'name': 'Salt', 'weight': '1kg', 'price': 112.8}
```

Python Dictionaries

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict["color"] = "red"
print(this_dict)
```

Python Dictionaries

Removing Items

There are several methods to remove items from a dictionary:

The pop() method removes the item with the specified key name:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict.pop("price")
print(thisdict)
```

Python Dictionaries

Removing Items

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict.popitem()
print(thisdict)
```

Python Dictionaries

Removing Items

The del keyword removes the item with the specified key name:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
del this_dict["price"]        print(this_dict)
```

The del keyword can also delete the dictionary completely:

```
del this_dict
print(this_dict) #this will cause an error because "thisdict" no longer exists.
```

Python Dictionaries

Removing Items

The clear() method empties the dictionary:

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
this_dict.clear()
print(this_dict)
```

Python Dictionaries

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a *reference* to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
my_dict = this_dict.copy()
print(my_dict)
```

Python Dictionaries

Copy a Dictionary

Another way to make a copy is to use the built-in function dict().

```
this_dict = {
  "name": "Sugar",
  "weight": "1kg",
  "price": 130.35
}
my_dict = dict(this_dict)
print(my_dict)
```

Python Dictionaries

Nested Dictionaries

Nested dictionaries in Python refer to dictionaries that are contained within another dictionary as values.

```
student_records = {
    "John": {
        "age": 20,
        "grades": {
            "math": 85,
            "science": 92,
            "history": 78
        }
    },
    "Emily": {
        "age": 22,
        "grades": {
            "math": 92,
            "science": 88,
            "history": 95
        }
    }
}
```

print(student_records[“John”][“age”]) # Output 20

print(student_records[“Emily”][“grades”][“science”]) # Output 88

What are Operators?

Operators in Python are special symbols that perform operations on variables and values. They are used to perform arithmetic operations, logical operations, and comparisons.

Here are the 7 main types of operators in Python:

Arithmetic Operators

Assignment Operators

Comparison Operators

Logical Operators

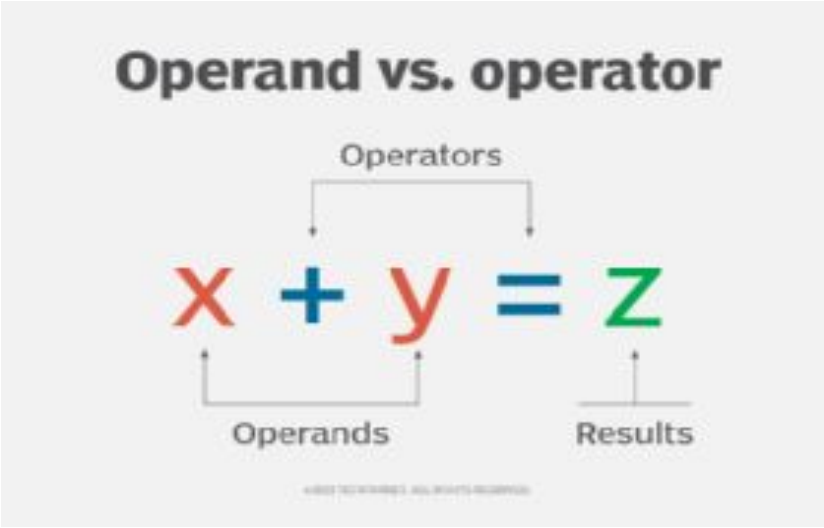
Identity Operators

Membership Operators

Bitwise Operators

Arithmetic Operators

These operators are used to perform mathematical operations on numbers.



Arithmetic Operators

The arithmetic operators return the type of result depends on the type of operands, as below.

- If **either operand is a complex number**, the result is converted to complex;
- If **either operand is a floating point number**, the result is converted to floating point;

Arithmetic Operators

Operation	Operator	Example in Python Shell
Addition	+	`x,y = 5,6` `print(x + y)`
Subtraction	-	`x,y = 5,6` `print(x - y)`
Multiplication	*	`x,y = 5,6` `print(x * y)`
Exponentiation	**	`x,y = 5,6` `print(x ** y)`
Division	/	`x,y = 5,6` `print(x / y)`
Floor division	//	`x,y = 5,6` `print(x // y)`
Modulus	%	`x,y = 5,6` `print(x % y)`

Assignment Operators

The assignment operators are used to **assign values to variables**.

A good practice

Note: According to **PEP 8** – Python's official style guide: "**Surround binary operators with a single space on both sides.**"

Assignment Operators

Operation	Operator	Example in Python Shell
assignment	=	`x = 5` `print(x)`

add and assign	+=	`x = 5`
		`x += 7`
		`print(x)`

subtract and assign)	-=	`x = 5`
		`x -= 7`
		`print(x)`

multiply and assign	*=	`x = 5`
		`x *= 7`
		`print(x)`

Assignment Operators

Operation	Operator	Example in Python Shell

divide and assign	/=	`x = 5`
		`x /= 7`
		`print(x)`

modulo and assign	%=	`x = 5`
		`x %= 7`
		`print(x)`

exponentiation and assign	**=	`x = 5`
		`x **= 7`
		`print(x)`

floor division and assign	//=	`x = 5`
		`x //= 7`
		`print(x)`

Comparison Operators

– – –

The comparison operators **compare two operands** and return a boolean either True or False.

Comparison Operators

Operation	Operator	Example in Python Shell
Equal to	==	`x,y = 5,6` `print(x == y)`
Not equal to	!=	`x,y = 5,6` `print(x != y)`
Less than	<	`x,y = 5,6` `print(x < y)`
Greater than	>	`x,y = 5,6` `print(x > y)`
Less than or equal to	<=	`x,y = 5,6` `print(x <= y)`
Greater than or equal to	>=	`x,y = 5,6` `print(x >= y)`

Logical Operators

Logical operators in Python are used to **perform logical operations on boolean values** (True or False) or expressions. These operators allow you to combine or manipulate boolean values to make more complex decisions or evaluations.

Logical Operators

Operation	Operator	Example in Python Shell
Logical AND	and	`x,y = 5,6` `print(x > 1 and y < 10)`
Logical OR	or	`x,y = 5,6` `print(x > 6 or y < 10)`
Logical Not	not	`x = 5` `print(not x > 1)`

Identity Operators

Identity operators in Python are used to **compare the memory locations (identities) of two objects**. They determine whether two variables or values refer to the same object in memory.

Identity Operators

— — —

Operation	Operator	Example in Python Shell
Is reference to the same object	is	`x,y = 5,6` `print(x is y)`
Is not reference to the same object	is not	`x,y = 5,6` `print(x is not y)`

Identity Operators

But wait, why below?

```
a = 5

b = 5

print(a is b)   # ✅ True
```

Identity Operators

— — —

But wait, why below?

```
a = 5

b = 5

print(a is b)   # ✅ True
```

In Python, small integers, specifically those in the range from -5 to 256 (inclusive), are pre-created and cached in memory when the Python interpreter starts. This is a performance optimization known as **integer interning** or the **small integer cache**.

Membership Operators

Membership operators in Python are used to **test whether a value or variable is a member of a sequence** (like a list, tuple, string, or set or even range) or not. These operators help you determine if a particular value exists within a given sequence.

Membership Operators

— — —

Operation	Operator	Example in Python Shell

Is reference to the same object	in	`nums = [1,2,3,4,5]`
		`print(1 in nums)`
		`print(10 in nums)`

Is not reference to the same object	not in	`nums = [1,2,3,4,5]`
		`print(1 not in nums)`
		`print(10 not in nums)`

Bitwise Operators

Bitwise operators in Python are used to **perform operations on the individual bits** of integer values. These operators allow you to manipulate and compare the binary representations of integers at the bit level.

Bitwise operations are often used in low-level programming, hardware manipulation, and certain algorithms that require bit-level control.

Bitwise Operators

Operation	Operator	Example in Python Shell

Bitwise AND	&	`x = 5  y = 10`
		`z = x & y`
		`print(z)`

Bitwise OR	|	`x = 5  y = 10`
		`z = x | y`
		`print(z)`

Bitwise XOR	^	`x = 5  y = 10`
		`z = x ^ y`
		`print(z)`

Bitwise NOT	~	`x = 5` `print(~x)`

Bitwise Operators

Operation	Operator	Example in Python Shell
`Left shift`	<<	`x = 5` `print(x << 2)`
`Right shift`	>>	`x = 5` `print(x >> 2)`

Operator Precedence

Operator precedence describes the order in which operations are performed.

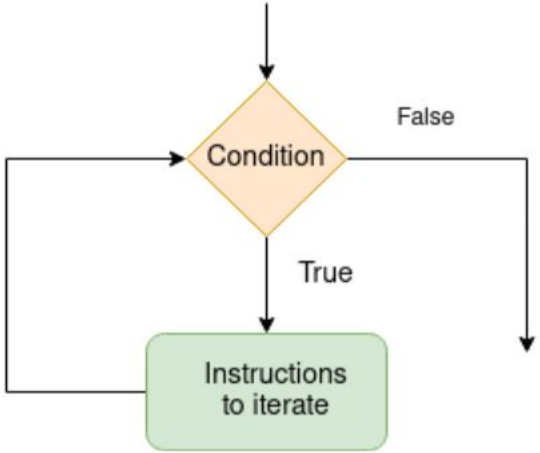
Level	Category	Operators
7(high)	exponent	**
6	multiplication	*,/,//,%
5	addition	+,-
4	relational	==,!=,<=,>=,>,<
3	logical	not
2	logical	and
1(low)	logical	or

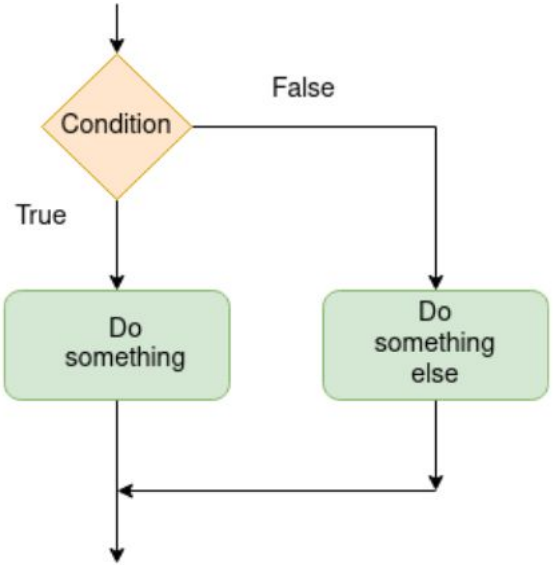
Control Flow Statements

What Control Flow is?

Control flow refers to **the order in which statements and instructions are executed** in a program.

What Control Flow is?





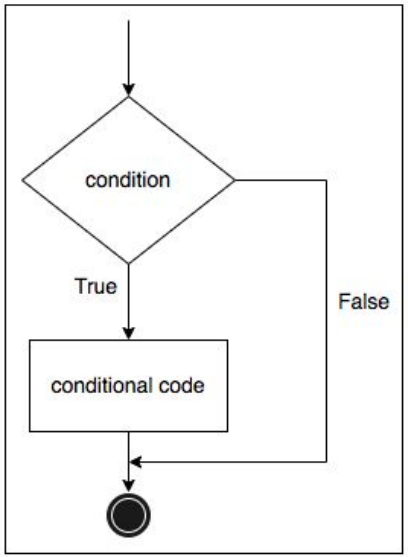
Python Control flow

What are Control Flow Statements?

Control Flow Statements are special **commands that change that order**, like making decisions or repeating actions.

Python if Statement

It help us to run a particular code, but only when a certain condition is met or satisfied. A **if** only has one condition to check.



Python if Statement

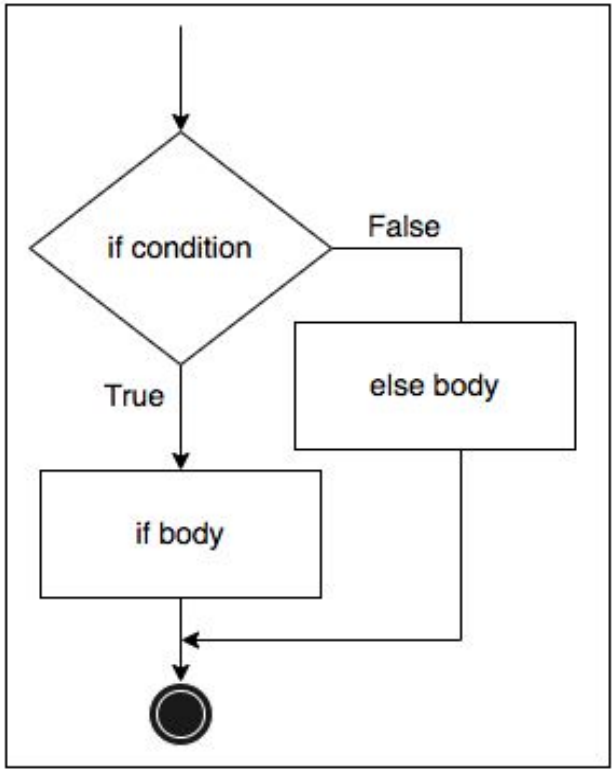
In the following code example, we're checking the if condition, e.g., if n modulo 2 equals to 0 then the print statement will execute.

```
1  n = 10
2  if n % 2 == 0:
3      print("n is an even number")
```

Python if-else Statement

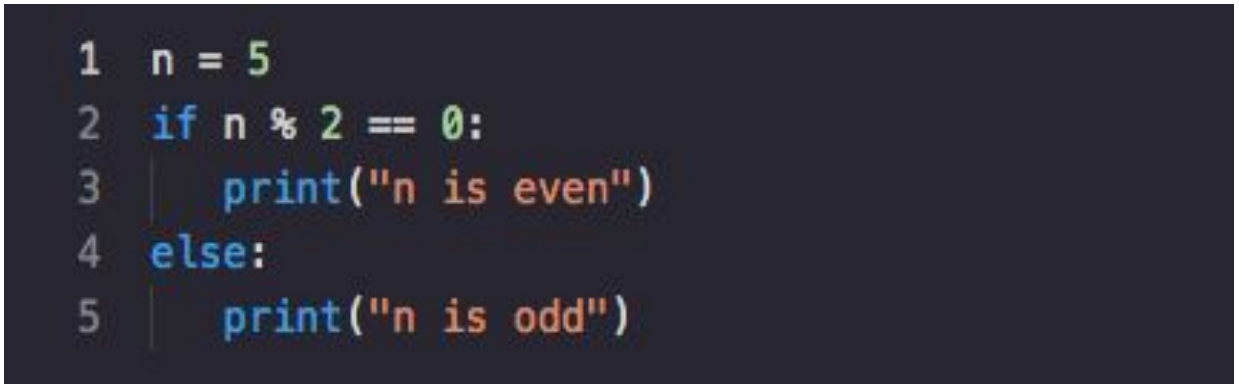
The **if-else** statement evaluates the condition and executes the body of the '**if**' block, if the test condition is true. However, if the condition is false, the body of the '**else**' block is executed.

Python if-else Statement

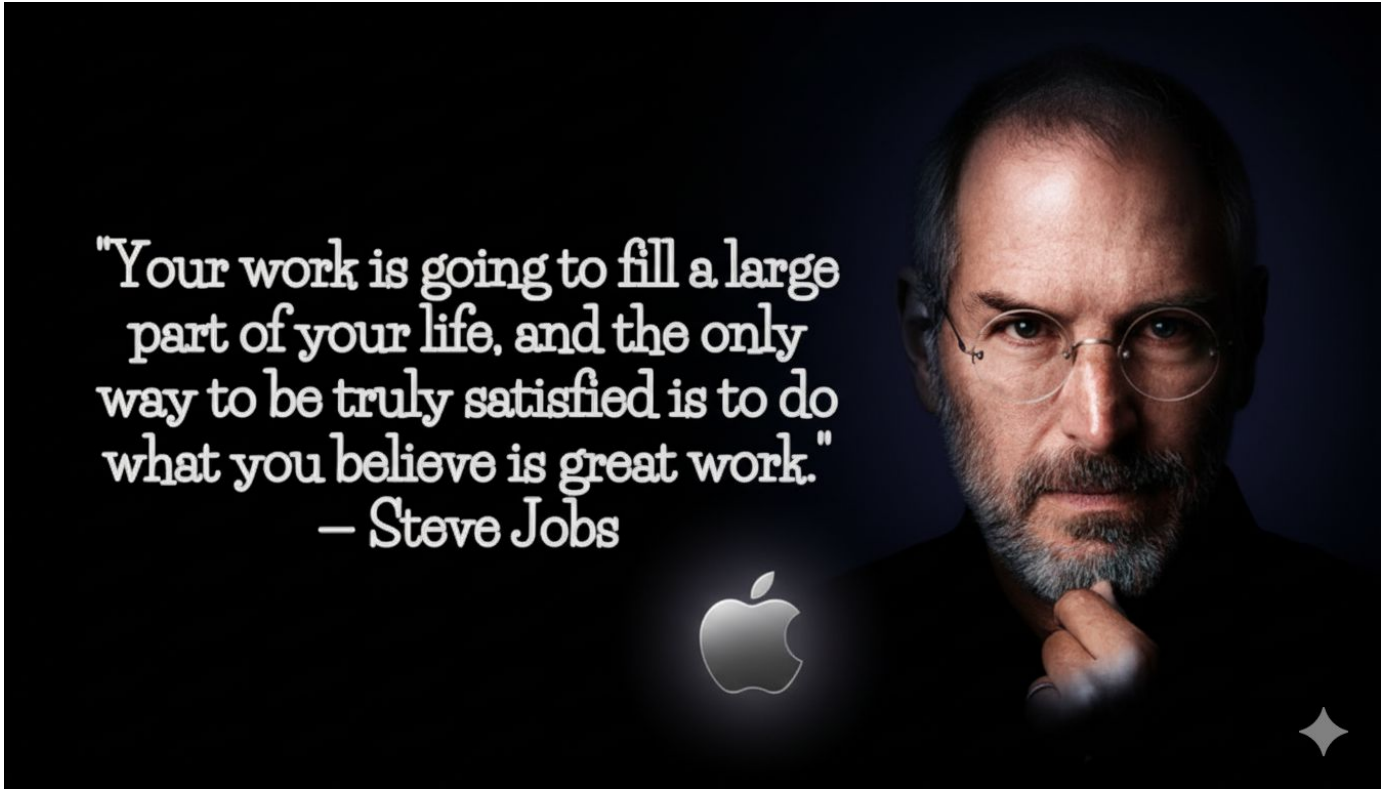


Python if-else Statement

In the following code example, the else body will execute as 5 modulo 2 is not equal to 0.



Thank You...!


"Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work."
– Steve Jobs